Semi-Supervised Log Pattern Detection and Exploration Using Event Concurrence and Contextual Information

Xixi Lu[1], Dirk Fahland[1], Robert Andrews[2], Suriadi Suriadi[2],
Moe T. Wynn[2], Arthur H.M. ter Hofstede[2], Wil M.P. van der Aalst[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Queensland University of Technology, Brisbane, Australia

Abstract. Process mining offers a variety of techniques for analyzing process execution event logs. Although process discovery algorithms construct end-to-end process models, they often have difficulties dealing with the complexity of real-life event logs. Discovered models may contain either complex or over-generalized fragments, the interpretation of which is difficult, and can result in misleading insights. Detecting and visualizing behavioral patterns instead of creating model structures can reduce complexity and give more accurate insights into recorded behaviors. Unsupervised detection techniques, based on statistical properties of the log only, generate a multitude of patterns and lack domain context. Supervised pattern detection requires a domain expert to specify patterns manually and lacks the event log context. In this paper, we reconcile supervised and unsupervised pattern detection. We visualize the log and help users *extract* patterns of interest from the log or obtain patterns through unsupervised learning automatically. Pattern matches are visualized in the context of the event log (also showing concurrency and additional contextual information). Earlier patterns can be extended or modified based on the insights. This enables an interactive and iterative approach to identify complex and concrete behavioral patterns in event logs. We implemented our approach in the ProM framework and evaluated the tool using both the BPI Challenge 2012 log of a loan application process and an insurance claims log from a major Australian insurance company.

1 Introduction

Process discovery offers automated construction of an end-to-end process model from an event log, recorded when executing a real-life business process, to gain useful insights into process executions. However, real-life logs often contain complex behavior which discovery algorithms struggle to handle. Consequently, discovered models may contain either very complex or over-generalized fragments, the interpretation of which is difficult. This can result in misleading insights. Fig. 1(a) exemplifies an event log which contains four traces over five activities; from the log, an existing discovery algorithm (e.g., the Inductive Miner [1]) may return an over-generalized model allowing all five activities to be executed or skipped in any order, as shown in Fig. 1(b).

Discovery algorithms struggle because they aim to represent all variants and contexts of an activity within one single model. Instead, the use of patterns to untangle these complex behaviors and address them separately, as well as the detection of these

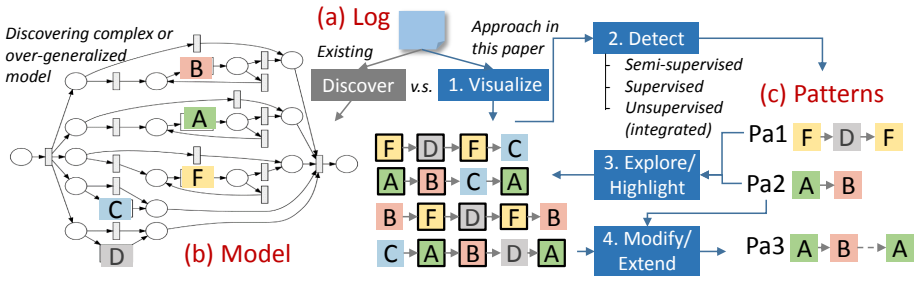

Fig. 1: An example of a process discovery problem versus the approach in this paper.

patterns on event logs, can uncover more accurate insights into the complex behavior in process executions [2–9]. However, existing approaches to pattern detection are either *unsupervised learning* or *supervised learning*, with each having their own limitations, as depicted by Fig. 2.

Unsupervised learning [3,7] generates patterns based on statistical properties of the log, such as frequency, support and confidence, and suffers from a problem known as "pattern explosion". Patterns that are less frequent are more difficult to detect. Moreover, to the best of our knowledge, no existing unsupervised approach in log pattern detection supports retrieving occurrences of patterns. In contrast, most *supervised approaches* [5,6] assume a pattern set will be given, e.g., manually drawn by the experts, which puts the burden on the modeler.

In this paper, we propose a semi-automated approach to detecting *log patterns* from an event log (where events of traces could be totally ordered or *partially ordered*); an overview of the approach is shown in Fig. 1. We define the log patterns as short acyclic process fragments (formally: *partial orders* of activities where direct and indirect succession of activities are specified) and the semantics of the patterns as instances detected in an event log. As shown in Fig. 1, the approach starts with (1) *visualizing* the traces in a dotted-chart-like manner. The visualization allows users to interactively (2) *detect* and *extract* log patterns of interest. The occurrences of a log pattern in traces can be (3) *highlighted* in the log visualization, to support users *exploring* patterns in their context and comparing occurrences of patterns. Users can (4) *modify* or *extend* patterns based on the occurrences (by extending them with additional activities, different ordering relations, etc.) or create new patterns based on existing ones. This way we enable the

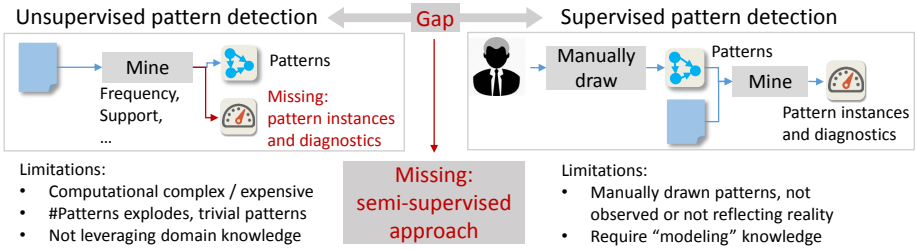

Fig. 2: Problem analysis: missing semi-automated pattern detection for log exploration.

user to explore the patterns in their context (where and how frequently they occur) and modify patterns interactively and iteratively, aiming to help the user to (i) explore an event log with ease, (ii) obtain more concrete and accurate insights into process behavior than is possible from some end-to-end discovered models, and (iii) balance between unsupervised and supervised learning.

We implemented the functionalities for all these four steps in a log visualization tool as a plug-in for the ProM process mining framework[3]. For step (2) detecting log patterns, we proposed three semi-automated detectors and integrated two existing unsupervised approaches (by converting the detected patterns into our log patterns). Moreover, we support users to convert traces into partial orders. The aim is to *complement* and *not* substitute existing pattern detection approaches. We evaluated the approach in two case studies conducted using real-life event data sets. The results show concrete insights into process behavior patterns which existing discovery algorithms cannot reveal.

The remainder of this paper is organized as follows. Sect. 2 discusses related work. Sect. 3 discusses the preliminaries and the input for our approach. In Sect. 4, we formally define log patterns and pattern instances. In Sect. 5, we explain our approach to detect patterns and find pattern instances. Sect. 6 reports results for the two case studies conducted. Sect. 7 concludes the paper and discusses future work.

2 Related work

We use Table 1 to discuss and structure related work. Here we discuss related work on log pattern detection on a conceptual level; an evaluation of the unsupervised techniques on a real-life event log is discussed in Sect. 6.

Log Pattern Definition. Patterns in the process mining literature have been defined in various ways. Early work focuses on clustering frequently co-occurring activities; such a cluster of activities is considered as a (low-level) pattern and mapped into a high level activity [4, 10]. Later, more specialized definitions are used. Bose et al. [3] defined patterns as repeated sequences. [7] and [8] defined patterns as partial orders of activities in which the edges only represent eventually-cause relations. The work in [5, 11] considered patterns as Petri nets. In our case, we use partial orders, distinguishing concurrence, directly-cause and eventually-cause.

Unsupervised Log Pattern Detection. Unsupervised log pattern detection approaches take an event log as input and generate patterns based on predefined measures [3, 7, 8]. This has some limitations. Firstly, such unsupervised approaches are computationally complex and expensive, generating a massive amount of possible patterns based on their frequencies or other measures. If one sets the values for the measures too high, then only very frequent, trivial patterns are returned, thereby missing many interesting results. By setting the values too low, too many patterns are returned (so called "pattern explosion") [3,7,8]. Secondly, as a result of not leveraging domain knowledge, many of the patterns generated by unsupervised learning are not of interest or are meaningless. Finally, most unsupervised approaches do not return pattern instances or additional con-

[3] http://www.promtools.org/

Table 1: Comparison of related pattern detection approaches.

	S/U/M/V*	provide unsuper. support	Change/ define patterns?	Explore pattern instances?
Bose et al., Pattern abstraction [3]	U	+ (beh.)†	-	-
Günther et al., Fuzzy mining [4]	U	+ (act.)	-	-
Mannhardt et al., Log abstraction [5]	S	- (act.)	+	+/-
Maggi et al., LTL checker [6]	S	- (com.)	+/-	+/-
Leemans et al., Episodes miner [7]	U	+ (beh.)	-	-
Diamantini et al., Pattern discovery [8]	U	+ (beh.)	-	-
Baier et al., Activity maching [12]	M	- (act.)	-	-
Ferreira et al., Label abstraction [10]	U	+ (act.)	-	-
Tax et al., Local models [11]	U	+ (beh.)	-	-
Song et al., Dotted chart [13]	V	- (beh.)	-	-
Shneiderman et al., EventFlow [2]	S.V.	- (beh.)	+	+
Lu et al., Pattern explorer	M.S.V.U.	+ (beh.)	+	+

* S. for supervised; U. for unsupervised; M. for Semi-Supervised; V. for visualization.
† In parentheses, the aim of the technique is abbreviated: beh. for behavior analysis; act. for low level activity abstraction; com. for compliance checking.

textual information for the detected patterns, thus obstructing the user from exploring and analyzing the patterns [7].

Supervised Log Pattern Detection. Supervised pattern detection approaches in process mining take patterns and logs as input and detect pattern instances as results [5]. Such supervised approaches require the user to model patterns in a formal language such as Petri nets or LTL constraints. This relies on the expertise of the user, who may need to formalize a large set of pattern descriptions. Moreover, the user may miss potentially important patterns through incomplete specification or model idealized patterns that are not observed in reality.

Log Explorer and Visualization. Advanced log visualization analytics have also been proposed as a way to help the user observe patterns. The dotted chart [13] is a simple way of visualizing event logs and helping the user spot and interpret patterns such as batch processing. However, no pattern extraction approaches are supported, nor is it possible to query for all instances of the observed patterns. EventFlow [2] has been proposed as a more advanced tool for visualizing event sequences. It allows for advanced querying, but also requires the user to create patterns (queries) and does not support generating patterns in an unsupervised or semi-unsupervised way. In our case, we allow semi-supervised pattern detection; the detected patterns are suggested to the user and can be used as queries. Moreover, we support partially ordered events and help the user detect and explore causal dependencies between events [14].

3 Preliminaries

In this section, we recall the basic concepts such as *partial orders*, *event logs*, and *partially ordered traces* and discuss the *oracle functions* that convert sequential traces of an event log into partially ordered traces. These are used later in the paper.

Let X be a set of elements and $X' \subseteq X$ a subset of X. A relation $R \subseteq X \times X$ between X is a *partial order* over X if and only if R is irreflexive, anti-symmetric, and transitive. Let $G = (N, <)$ be a *directed acyclic graph (DAG)* with $< \subseteq N \times N$. We use $<^-$ to denote transitive reduction of $<$ and $<^+$ to denote the transitive closure of $<$. The relation $<^+$ will be used to denote reachability and defines a partial order. In this paper, for all nodes $n, n' \in N$, $n \not<^+ n'$ and $n' \not<^+ n$, we say n and n' are *concurrent* and use $n \;||_< \; n'$ to denote this. We use $\downarrow$ to denote a *projection function*, i.e., $X \downarrow_{X'} = X \cap X'$ and $R \downarrow_{X'} = R \cap (X' \times X')$. Let $G = (N, <)$ be a DAG and $N' \subseteq N$. We overload the projection function and define the projection for a graph, $G \downarrow_{N'} = (N \downarrow_{N'}, <\downarrow_{N'})$, also known as *induced subgraph*.

An *event log* represents the observed behavior of a process. Each case going through the process results in a trace of events in the event log.

Definition 1 (Event, trace, event log). *Let $\mathcal{E}$ be the universe of unique* events*, i.e., the set of all possible event identifiers. A* trace $\sigma = \langle e_1, \cdots, e_n \rangle \in \mathcal{E}^*$ *is a finite sequence of events $e_1, \cdots, e_n \in \mathcal{E}$. An* event log $L = \{\sigma_1, \sigma_2, \cdots, \sigma_n\} \subseteq \mathcal{E}^*$ *is a set of traces.*

Here we assume that no event appears twice in the same trace nor in the same log. We use E_σ for the set of events in trace σ and E_L for the set of events in log L. We use $\pi_{act}(e)$ to return the activity associated with e; and $\pi_{time}(e)$ to return the timestamp of e. In this paper, we use π_{act} as our default labeling function for events and assume that both π_{act} and π_{time} are universally available for $\mathcal{E}$. Fig. 3(a) shows an example of totally ordered trace $\sigma = \langle e_1, e_2, ..., e_5 \rangle$. Event e_1 has activity $\pi_{act}(e_1) =$ *Injury* and is executed on $\pi_{time}(e_1) =$ 08/09/2016-00:30:00.

Many recent papers consider partial orders of events [14–16], instead of totally ordered event sequences. One reason for this consideration is that a particular total order of events may be unreliable or unknown. For example, if events a and b are recorded only on day granularity (not seconds), then a totally ordered log may contain the sequence $\langle a, b \rangle$ whereas in reality $\langle b, a \rangle$ occurred. Representing events as a partial order (where a and b can occur "unordered" or "concurrent") alleviates this problem and allows us to represent more accurate contextual information of events [14, 16].

Definition 2 (Partially Ordered Trace). *A* partially ordered trace $\varphi = (E, \prec)$ *is a Directed Acyclic Graph (DAG), in which $\prec$ denotes the inferred* "cause" relations[4] *over events E. If $e \prec e'$, we say e* caused *e'. We use $\prec^-$ to denote* directly-cause*, $\prec^+$ to denote* eventually-cause*, and $||_\prec$ to denote the* concurrent *relation.*

Note that the *eventually-cause* relation $\prec^+$ forms a partial order. Fig. 3(b) shows a partially ordered trace $(E, \prec)$, in which $E = \{e_1, e_2, \cdots, e_5\}$ and $\prec = \{(e_1, e_2), (e_1, e_3), (e_2, e_4), (e_3, e_4), (e_4, e_5)\}$. In this particular case, $\prec^- = \prec$ and $\prec^+ = \prec \cup \{(e_1, e_4), (e_1, e_5), (e_2, e_5), (e_3, e_5)\}$. The concurrence relation $||_\prec = \{(e_2, e_3), (e_3, e_2)\}$. Note that $\prec$, $\prec^-$, and $\prec^+$ are irreflexive and acyclic, and $||_\prec$ is irreflexive and symmetric.

Conversion. Partial orders of events may be obtained from totally ordered traces. Some approaches [14, 15, 17] assume to have an oracle that indicates the set of activities that are concurrent or unordered and use this oracle to convert totally ordered events

[4] We use the term *"cause" (causality)* only to distinguish the relations in a partially ordered trace from the *follow* relations (i.e., *directly-follow* and *eventually-follow*) in totally ordered traces.

into partial orders. Such an oracle could be obtained by interviewing domain experts or computed from logs [14].

We overload the symbol φ to denote an *conversion oracle function* that, for a trace σ, returns a partially ordered trace $\varphi(\sigma) = (E_\sigma, \prec_\sigma)$. We deploy the oracle that considers the events occurring within a short time to be concurrent [14] as our default oracle. Let $\sigma = \langle e_1, \cdots, e_n \rangle$ be a trace. $\varphi_{time}(\sigma, dt) = (E_\sigma, \prec_\sigma)$ in which $\prec_\sigma = \{(e_i, e_j) \mid 1 \leq i < j \leq n \wedge \exists_{i \leq k < j} \pi_{time}(e_{k+1}) - \pi_{time}(e_k) \geq dt\}$. In addition, we also overload the φ function to handle a log $L = \{\sigma_1, \cdots, \sigma_n\}$ and return a set of partially ordered traces, one for each trace in L, i.e., $\varphi(L) = \{\varphi(\sigma_1), \cdots, \varphi(\sigma_n)\}$. An example of such a conversion based on the timestamps is shown in Fig. 3, with $dt = 1.0$ sec.

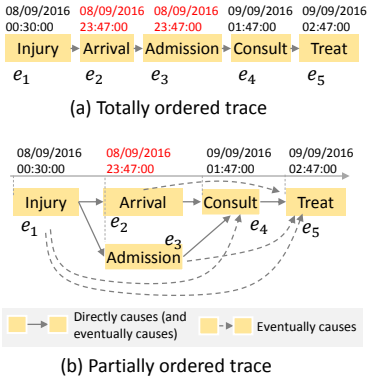

Fig. 3: A sequential trace and its converted partially ordered trace.

4 Patterns and Pattern Instances

Having defined partially ordered traces of an event log, we now present the concept of log patterns. In Sect. 4.1 we motivate and define the log patterns and pattern instances. In Sect. 4.2, we discuss pervasiveness measures for the patterns, such as support, confidence and coverage. In Sect. 5, three approaches to pattern detection are presented.

4.1 Core-Activity, Pattern and Pattern Instance

To support process analysts in expressing and modifying log patterns with ease, the patterns should be simple. Moreover, if such a pattern resembles our input traces, it would be easier for the user to observe and recognize these patterns. We therefore define a pattern as a labeled DAG (similar to a partially ordered trace) and allow a pattern to express causal dependencies that occur in a partially ordered trace, namely *directly-cause* and *eventually-cause*. The *concurrence* relation is then deduced for any two events that are not eventually causing one another. Fig. 4(a) and (d) show two examples of patterns.

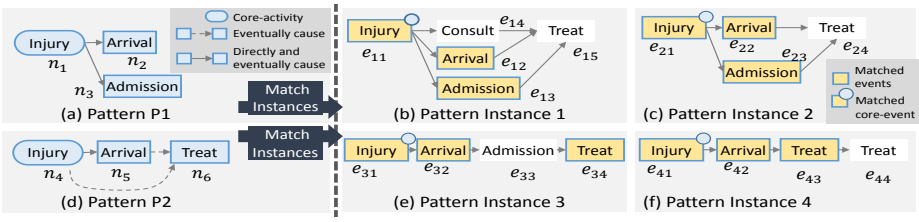

Fig. 4: Two patterns and four highlighted pattern instances, two for each pattern.

In addition to considering patterns as labeled DAGs, we, for two reasons, explicitly include a notion called a *core-activity* in the patterns. Firstly, the *core-activity* notion allows for unambiguous notions of pattern matches and unambiguous computation of

measures such as frequency. Without a core-activity, having a pattern P_{ab} that says "a eventually-caused b" and a trace $\sigma_1 = \langle a, b, b, a \rangle$, should this be considered as one instance or two? Given another trace $\sigma_2 = \langle a, b, a, b, c \rangle$, if the number of combinations is considered, such an approach results in three pattern instances in σ_2 while there are only two a's and two b's, causing confusion. In our case, the core-activity anchors the pattern in a particular perspective. We simply count the number of distinct events that match the core-activity and satisfy the pattern and call these events *core-events*; we ignore whether the events that match to the other nodes in a pattern occur several times. Take the same example, if a is the core-activity of P_{ab}, then we have one a for σ_1 and two a's for σ_2. Similarly, if the pattern P_{ab} has b as the core-activity, then we have two b's for σ_1 and also two b's for σ_2. In addition, having a core-activity also allows to unambiguously compute the events (*anti-instances*) that do *not* satisfy a pattern (not just the traces). For example, having the same pattern P_{ab} with a as the core-activity, the second a in σ_1 does not satisfy the pattern; without the core-activity, σ_1 may be consider as compliant or not, depending on the interpretation of the pattern.

Definition 3 (Log Pattern). *A log pattern $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ is a directed acyclic graph, where:*

- *N is a set of nodes,*
- *$\mapsto$ is a set of edges among N and denotes the directly-cause relation[4],*
- *$\rightsquigarrow$ is a set of edges among N and denotes the eventually-cause relation[4],*
- *$\alpha : N \to A$ is a partial function that assigns a label $\alpha(n)$ to any node $n \in N$,*
- *$c \in N$ is the core-activity of the pattern*

and satisfies the following constraints:

1. *$\mapsto \subseteq \rightsquigarrow$, i.e., the directly-cause relation is a subset of the eventually-cause relation;*
2. *$(N, \rightsquigarrow)$ is a partial order from which the concurrence $||_{\rightsquigarrow}$ can be deduced;*
3. *for all $n, n' \in N$, if there is $n'' \in N$ such that $n \rightsquigarrow n'' \rightsquigarrow n'$, then there is no $n \mapsto n'$. We also say c has context P and call $N \backslash \{c\}$ the* context-nodes *of c.*

Note that Def. 3 only defines the syntax of a log pattern. The semantics of a log pattern is defined by its instances in Def. 4. A pattern may be regarded as a core-activity (activity) that occurred in a particular context. Accordingly, an instance of the pattern is an occurrence of the core-activity in the log in the same context.

Definition 4 (Pattern Instance and Pattern Trace). *Let $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ be a log pattern, $\varphi = (E_\varphi, \prec)$ a partially ordered trace, $E' \subseteq E_\varphi$ a subset of events, and $e_c \in E'$ an event. (e_c, E') is an* instance *of pattern P if and only if there is a bijective function $I : E' \to N$ such that*

- *e_c (called the* core-event *of (e_c, E')) is mapped to the core-activity, i.e., $I(e_c) = c$;*
- *for each event $e \in E'$, event e and the corresponding node $I(e)$ have the same label, i.e., $\pi_{act}(e) = \alpha(I(e))$;*
- *for all events $e, e' \in E'$, the relations between e and e' satisfy the relations between $I(e)$ and $I(e')$, i.e., $(1)\ I(e) \mapsto I(e') \Rightarrow e \prec^- e'$, $(2)\ I(e) \rightsquigarrow I(e') \Rightarrow e \prec^+ e'$, and $(3)\ I(e') \ ||_{\rightsquigarrow}\ I(e) \Rightarrow e' \ ||_{\prec}\ e$*

If (e_c, E') is an instance of P, we also say e_c satisfies *P and say φ is a* trace *of P.*

The behavior specified by a pattern is preserved in the instances of the pattern. Fig. 4(b), (c), (e) and (f) exemplify four pattern instances highlighted in their partially

ordered traces: highlighted e_{11} and e_{21} satisfy P1; e_{31} and e_{41} satisfy P2. It is important to note that changing the core-activity of a pattern does not change the behavioral relations of the pattern, but does change the instances that match the pattern.

Definition 5 (A Maximal Set of Pattern Instances). *Let $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ be a pattern, L an event log, φ the conversion oracle. A maximal set of pattern instances $PI(P, \varphi(\sigma))$ of trace $\sigma \in L$ is defined as a maximal set of all instances of P in $\varphi(\sigma)$ that differ in their core-event, i.e., for any instance (e', E') of P, if $(e', E') \notin PI(P, \varphi(\sigma))$ then there exist $(e', E'') \in PI(P, \varphi(\sigma))$. We write $PI(P, L, \varphi) = \bigcup_{\sigma \in L} PI(P, \varphi(\sigma))$ for the union of a* maximal set of pattern instances *of all traces in log L. We write $PIC(P, L, \varphi)$ to denote* the set of all core-events *that satisfy P, i.e., $PIC(P, L, \varphi) = \{e \mid (e, E') \in PI(P, L, \varphi)\}$.*

The set of *anti-pattern instances* $AntiPIC(P, L, \varphi)$ is defined as the set of events that do not satisfy P, i.e., $AntiPIC(P, L, \varphi) = \{e \in E_L \mid \pi_{act}(e) = \alpha(c)\} \backslash PIC(P, L, \varphi)$. Note that independent of which maximal set of pattern instances is returned, the set of all core-events and the set of anti-pattern instances of a pattern remain the same. Consider for example pattern P2 in Fig. 4(d) and the four partially ordered traces shown on the right-hand side to be the set of all partially ordered traces in $\varphi(L)$. A maximal set of pattern instances of P2 in $\varphi(L)$ always contains four pattern instances with e_{11}, e_{21}, e_{31} and e_{41} as the core-events that satisfy P2. As e_{41} (*Injury*) in Fig. 4(f) already satisfies P2 with context-event e_{43} (*Treat*), e_{44} (*Treat*) is not considered as a context event. However, if n_6 (*Treat*) was considered as core-activity of P2, we would obtain five instances with the core-events $e_{15}, e_{24}, e_{34}, e_{43}$, and e_{44}. Furthermore, e_{31} and e_{41} are anti-pattern instances of pattern P1.

4.2 Pattern Support, Confidence and Coverage

To help the user assess the pervasiveness of a pattern, we define the following five measures of a pattern based on the set of all pattern instances. Let $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ be a pattern. Given a log L and the partially ordered traces $\varphi(L)$, we have the set of all core-events $PIC(P, L, \varphi) = \{e_1, e_2, \cdots, e_n\}$ that satisfy P.

- *Pattern support* indicates how many distinct events satisfy P. i.e., $P\text{-}supp(P, L, \varphi) = \mid PIC(P, L, \varphi) \mid$.
- *Pattern confidence* is the number of events that satisfy P divided by the total number of events that have the same label as the core-activity; this measure indicates how often is the occurrence of the core-activity a predictor for the occurrence of the entire pattern. i.e., $P\text{-}conf(P, L, \varphi) = \frac{\text{P-supp}(P,L,\varphi)}{|\{e \in E_L | \pi_{act}(e) = \alpha(c)\}|}$.
- *Case support* is the number of traces that have at least one pattern instance satisfying P, i.e., $C\text{-}supp(P, L, \varphi) = \mid \{\sigma \in L \mid \exists\, e \in PIC(P, L, \varphi), e \in E_\sigma\} \mid$.
- *Case confidence* is the case support of P divided by the number of cases that have an event with the same label as c, i.e., $C\text{-}conf(P, L, \varphi) = \frac{C\text{-}supp(P,L,\varphi)}{|\{\sigma \in L | \exists\, e \in \sigma, \pi_{act}(e) = \alpha(c)\}|}$.
- *Case coverage* is the case support of P divided by the number of cases in the log, i.e., $C\text{-}cover(P, L, \varphi) = \frac{C\text{-}supp(P,L,\varphi)}{|L|}$.

We note that the desirability of pervasiveness measures (highly pervasiveness or otherwise) is context/application dependent. For example, for patterns representing non-complaint behavior or data quality issues, we would prefer to see low pervasiveness.

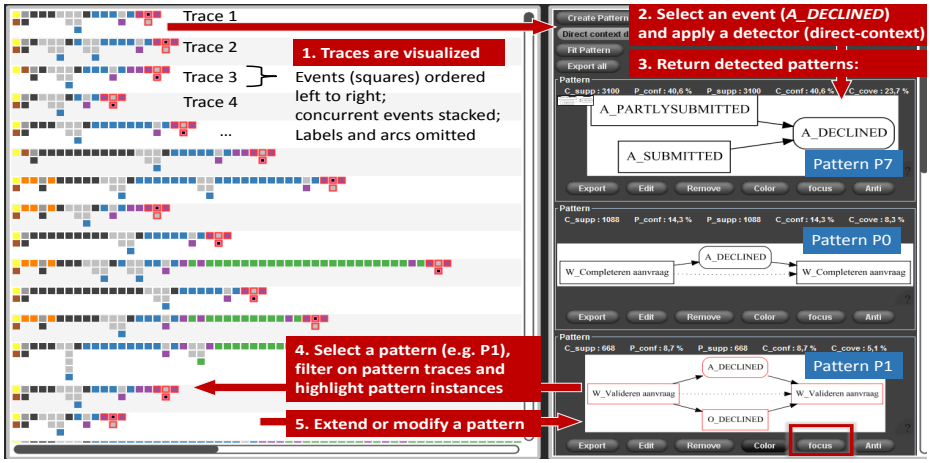

Fig. 5: Snippet of the tool after it found 9 patterns using event *A_DECLINED* and filtered on the pattern traces of pattern P1.

5 Pattern Detection and Pattern Instance Matching

We now introduce operations to *extract* and *extend* patterns in the context of a log in an explorative approach. To help the reader envision this explorative approach, we first briefly introduce the *Log Pattern Explorer (LPE)* tool, for which a screenshot is shown in Fig. 5. The right-hand side panel shows the log patterns, manually or automatically extracted. The user may create or modify a pattern. On the left-hand side, each log trace is visualized as a partial order in its own panel: each event is visualized as a square tile; tiles can be colored based on event attributes; concurrent events are stacked on top of each other; the labels and arcs are omitted for the sake of simplicity. When a user selects one or more patterns in the right-hand panel, all pattern instances are highlighted on the left (by a color-coded frame around the tiles of the satisfying events).

In Sect. 5.1, we discuss various ways to extract, detect, and modify patterns using supervised, semi-supervised or unsupervised approaches. We then discuss an approach to compute a maximal set of pattern instances of a pattern (Def. 5) in Sect. 5.2.

5.1 Pattern Detection Approaches - Partially Ordered Traces To Patterns

Definition 3 and the tool allow the user to create any pattern of interest. Nevertheless, as shown in Fig. 1, we would like to support the user in detecting these patterns from a log with ease. For example, an expert glances through the partially ordered traces visualized in Fig. 5 and may observe some events (patterns) reoccur in many traces, e.g., through the color coding. The user can then mark all events (tiles) in a trace that make up the pattern. An automated operation is defined in the following for extracting the marked events and their relations from the trace to build a complete pattern definition.

Pattern Extraction from Partially Ordered Traces. In essence, the marked events in a partially ordered trace φ are used to extract a subgraph of φ as the pattern. Formally, let $\varphi = (E, \prec)$ be a partially ordered trace, $E' \subseteq E$ a set of events marked by the user

and $e_c \in E'$ a chosen core-event. The function $extractPattern(\varphi, e_c, E') = P$ returns the log pattern P of E' induced on φ, i.e., $extractPattern(\varphi, e_c, E') = P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ where $N := E'$, $\mapsto := (\prec^-)\downarrow_{E'}$, $\rightsquigarrow := (\prec^+)\downarrow_{E'}$, $\alpha := (\pi_{act})\downarrow_{E'}$, and $c := e_c$.

Fig. 6 (Step 1) shows extraction of a pattern from the partially ordered trace by marking events $e_1(Injury)$, $e_2(Arrival)$ and $e_3(Admission)$ and considering e_1 as the core-event. The extracted pattern $P1 = (N, \mapsto, \rightsquigarrow, \alpha, c)$ with $N = \{n_1, n_2, n_3\}$, $n_1 \mapsto n_2, n_1 \mapsto n_3$, $n_1 \rightsquigarrow n_2$, and $n_1 \rightsquigarrow n_3$; n_2 and n_3 are concurrent; for α, the labels of nodes remain unchanged; the core-activity c is n_1 (*Injury*), abstracted from the core-event e_1. In the tool shown in Fig. 5, the user can mark a set of tiles (events) on the left and apply the pattern extraction function, the first tile is abstracted as the core-activity; the extracted pattern will appear in the panel on the right.

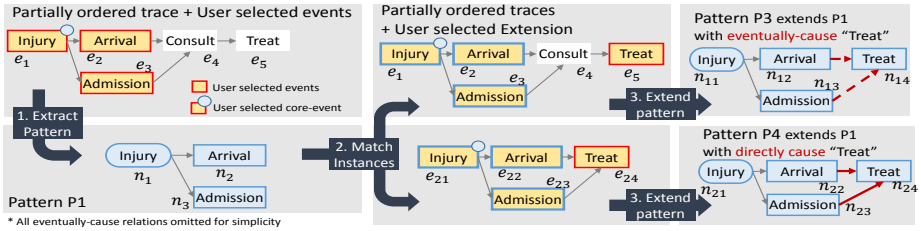

Fig. 6: Iteratively extracting and extending patterns from partially ordered traces.

Semi-Supervised Pattern Detection. To help the user detect patterns of interest, we also present three detectors that identify patterns for a user-chosen activity: (1) *concurrence detector*, (2) *direct-predecessor/successor detector*, and (3) *direct-context detector*. (1) The *concurrence detector* obtains a set of distinct patterns that describe different sets of activities which occurred concurrently to the user-chosen activity. Formally, let $\mathcal{P}$ be the set of patterns we have detected so far. Let L be a log, φ a conversion oracle, and c a core-activity of interest. Let $E_c = \{e \in E_L \mid \pi_{act}(e) = \alpha(c)\}$ be candidates for core-events. For event $e \in E_c$, let φ be the trace containing e, and let C_e be the events that are concurrent with e in φ. If $(e, C_e \cup \{e\})$ is not an instance of any pattern $P \in \mathcal{P}$, then we obtain a new pattern $P' = extractPattern(\varphi, e, C_e \cup \{e\})$ and add P' to $\mathcal{P}$. For (2) the *direct-predecessor (successor) detector*, C_e is the set of events that are directly-causing (that directly-caused by) core-event e. For (3) the *direct-context detector*, C_e contains directly preceding, succeeding and all concurrent events of e. The relevance of an extracted pattern can be assessed using the measures of Sect. 4.2.

Integrating Unsupervised Pattern Detection. To also leverage on existing unsupervised pattern detection techniques, their output has to be converted to our pattern notion (Def. 3). We discuss this conversion for two techniques [3, 7]. For the technique in [3], the output patterns are subsequences of activities. Any such pattern also satisfies our pattern definition (Def. 3). However, events that originally were totally ordered may now be independent (concurrent) due to the usage of partially ordered traces and may therefore no longer satisfy the converted pattern. In such cases, the user may find low *P-supp* and *P-conf*, explore anti-pattern instances and modify the pattern accordingly. The output patterns in [7] are partial orders of events in which the relations represent eventually-follow and no relations represent co-occurrence. We retain all eventually-

follow relations and choose to consider co-occurrence as concurrent. Regarding choosing the core-activities, the user may specify an activity (label) of interest to be automatically selected; otherwise, a random node is selected. The user can run such an unsupervised detection in the tool shown in Fig. 5. The returned and converted patterns are shown in the right panel. The user can explore the pattern instances in the left panel. Note that the pervasiveness of a pattern (such as *P-supp* and *P-conf*) are recomputed in our case, depending on the chosen core-activity. To reduce the efforts of manually converting patterns obtained from other approaches that are not integrated, we can standardize a file format (e.g., XML) for the patterns to import and exchange patterns.

Creating, Extending and Changing Patterns. Seeing the detected patterns and all instances of patterns in their larger context, the user may change a current pattern or create a new one. Def. 3 allows to check whether the new pattern is syntax-correct, and the algorithm that computes the instances of the pattern updates the semantics of the pattern, discussed in Sect. 5.2. Step 3 (extend pattern) in Fig. 6 exemplifies two different extensions of P1 that differ in how the added node *Treat* is relates to its predecessors.

5.2 Computing a Maximal Set of Pattern Instances

Having patterns obtained using one of the aforementioned methods, the semantics and the relevance of a pattern are defined by its pattern instances, as discussed in Sect. 4.1. To compute a *maximal set of instances* of a pattern (Def. 5), we present the following method, divided into three phases. First, all events that can be matched to core-activity c of pattern P are computed; we call these events the candidates of c and use E_c to denote this set of events. In the second phase, for each candidate $e \in E_c$, we try to find a pattern instance for P with e as core-event. This is done through an incremental construction of the mapping I (Def. 4) with backtracking and pruning. If we can complete the construction of I mapping to events E' in the trace, then (e, E') is a pattern instance and added to the maximal set. Else, e is an anti-pattern instance. The formal algorithm for computing the pattern instances is listed in [18]

The running-time complexity is exponential w.r.t. the size of the pattern (i.e., $|N|$), but polynomial in the size of E_c. In the best case, we try one combination and it already is an instance, then the algorithm runs in linear time. In the worst case, one may have to try every combination, then the algorithm runs in exponential time. However, we can incrementally check the validity of the chosen candidates so far through the projection function and efficiently prune the search space. Moreover, note that by only searching for a maximal set of pattern instances instead of all pattern instances, we evade exploring exponentially many matches for the same core-activity.

6 Evaluation and Discussion

The *Log Pattern Explorer (LPE)* is implemented as a log visualizer in the *LogPatternExplorer* package in the ProM framework[3]. We conducted two case studies. As discussed in Sect. 1, one objective is to evaluate whether (i) the semi-supervised approach supports the user in exploring log behavior and detecting complex patterns of interest. Another objective is to evaluate that (ii) the exploration of the occurrences of patterns

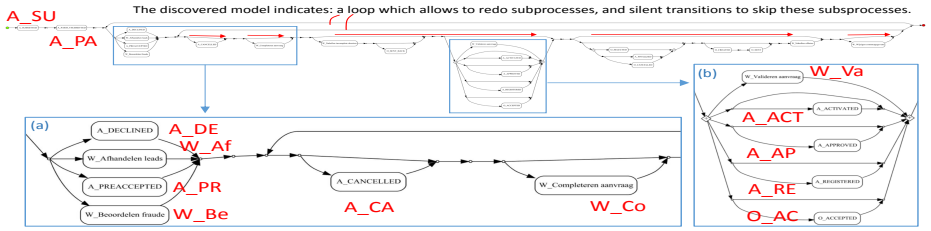

Fig. 7: The process model discovered on the BPI challenge 2012 log using IM; the model allows a large subprocesses to be executed in any order and any number.

in the log context helps to gain important and concrete insights into process behavior, especially on the parts where discovery algorithms have difficulties with. Furthermore, using some representative patterns detected in the first case study, the semi-supervised approach is compared to the existing unsupervised approaches to show that (iii) the proposed approach complements the existing unsupervised approaches by detecting complex, infrequent patterns and anti-pattern instances.

Guidelines for Exploring Log Patterns. The results in the case studies were obtained according to the following guidelines, using the tool described in Sect. 5 for steps 3-6.

1. A-priori, a process model was discovered using an existing discovery algorithm (e.g., the Inductive Miner [1]).
2. In the discovered model, unstructured parts may be detected (e.g., flower loops or activities with many or no connections). To understand these complex subprocesses and obtain more insights, the involved activities are used as a starting point for further analysis.
3. For each activity of interest, a first set of patterns (where the core-activity is labeled with the activity) is detected automatically, for instance, using the direct-context detector, to reveal different contexts (in the log) in which the activity occurs.
4. With each pattern detected, the corresponding pattern instances and pattern traces may be explored. A patterns is *interesting*, for example, if the pattern or the traces where the pattern occurred show behavior different from the discovered model.
5. During exploration, some detected patterns and their pattern traces may immediately show distinctive behaviors compared to the discovered model; some patterns may reveal similar behaviors and be grouped together; other patterns and their traces may be compared and reveal significant difference in their behavior.
6. During exploration, the same pattern may occur in distinct contexts (other events "around" the pattern). In such a case, the pattern is extended or modified to new patterns. The pattern traces of the new patterns are compared to gain more insights.

Evaluation using BPI Challenge 2012 Log. The BPI Challenge 2012 event log[5] was recorded for a loan application process in a Dutch financial institute. There are 13,087 cases in the log having in total 262,200 events and 24 activities and 4,366 distinct process instances.

Applying the Inductive Miner [1] with default settings (noise threshold: 0.2) on the BPI challenge log, we obtained a model shown in Fig. 7. This model indicates that, af-

[5] 10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

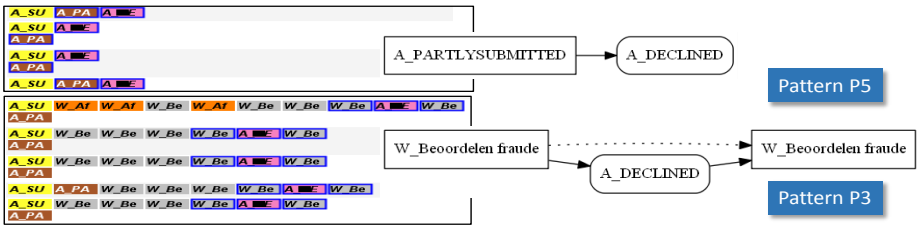

Fig. 8: Patterns P5 and P3 and their *pattern traces* show simple behavior, different from the discovered model in Fig. 7 and different from each other.

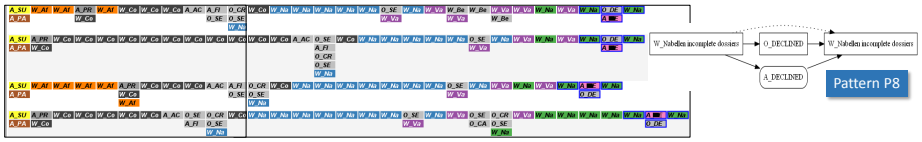

Fig. 9: Pattern P8 and its pattern traces which show different and more complex behavior when compared to the pattern traces of P3 and P5 as shown in Fig. 8.

ter the two activities, *A_SUBMITTED (A_SU)* and *A_PARTLYSUBMITTED (A_PA)*, all other activities are in a loop and all of them can be skipped (may be executed repetitively in any order) as highlighted in Fig. 7. The allowed behavior of this model is therefore rather general and does not help to understand the specific behavior of the process exhibited by the log, even though the model appears to be structured.

Starting from the discovered model, we select activities from the two submodels (a) and (b) highlighted in Fig. 7 and apply the LPE. We start with the four activities *A_DECLINED (A_DE)* , *W_Afhandelen leads (W_Af)*, *A_PREACCEPTED (A_PR)*, and *W_Beoordelen fraude (W_Be)* which according to the model may directly occur after the *A_PA* and can be followed by any other activity. We used the default oracle $\varphi_{time}(L, dt)$, with $dt = 1.0$ sec (e.g., events are concurrent if they happened within a second), to obtain partially ordered traces. We discuss our main findings [6].

Results regarding *A_DECLINED (A_DE)*. The direct-context detector returned 9 patterns P0, ..., P8 for the core-activity *A_DE*, indicating that *A_DE* occurs in 9 distinct contexts which we could group into three variants as follows. (1) The pattern traces of pattern P5 (and P7) show that when *A_PA* directly-causes *A_DE*, the process ends immediately after *A_DE* (in 3429 cases, 44.9% of all cases), exemplified in Fig. 8; no exceptional case was found. Thus, *A_DE* directly-preceded by *A_PA* does not occur repeatedly in a large subprocess (as suggested by the model of Fig. 7) but always at the end of a trace (as seen by the patterns in Fig. 8). (2) The pattern traces of P0, P2 and P3 show that *A_DE* occurs between repeated occurrences of the same activities (*W_Co*,*W_Af*, *W_Be*; Fig. 5 shows P3); all traces where P0, P2, P3 occur show the same more general pattern: the cases start with the same sequence of activities, only 8 out of the 24 process activities occurred, and each case is declined with P0, P2, or P3. (3) P1, P6, P8 show *A_DE* occurs concurrently to *O_DECLINED* with various predecessors/successors; traces containing these patterns are more complex and contain

[6] The resuls and detected patterns can be downloaded at prom-svn://Documentation/LogPatternExplorer

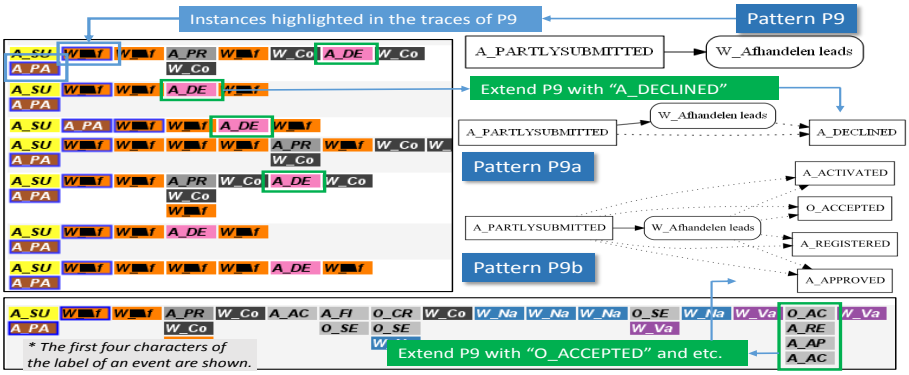

Fig. 10: The traces where pattern P9 occurs show a significant number of *A_DECLINED (A_DE)*; P9 is extended with eventually-cause *A_DE*.

activities (including "acceptance" activities) that do not occur in the cases discussed in (1) and (2); see Fig. 9. Altogether, we could identify 3 distinct variants. Moreover, where the model suggests complex behavior, such as the large sub-process after *A_DE* that allows complex behavior and can be repeated, the detected patterns show that all traces of these 9 patterns immediately ended after *A_DE* (plus one more activity).

Results regarding *W_Af* and *A_PR*. Following the same procedure, we investigated activities *W_Af*, and *A_PR*. We found 40 patterns (direct-contexts) for *W_Af* (of which 12 patterns have C-cover higher than 1%) and 9 patterns for *A_PR*. Exploring the pattern traces, we found two identical contexts for *W_Af* and *A_PR*: (P9) *W_Af* preceded by *A_PA* and (P10) *A_PR* preceded by *A_PA*. Visual inspection of the occurrences of P9 and P10 on the event log shows that *A_DE* occurs more frequently when P9 occurs than when P10 occurs. To confirm this, we extended P9 and P10 with "... eventually causes *A_DE*" to patterns P9a and P10a, respectively, as described in Sect. 5.1 (see Fig. 10). The relative share of the extended patterns $\frac{C-supp(P9a)}{C-supp(P9)} = 67.6\%$ and $\frac{C-supp(P10a)}{C-supp(P10)} = 19.4\%$ confirms our observation. Similarly, we observe that P9 "...eventually causes *O_ACCEPTED (O_AC)*" in 12% of the cases whereas this holds for P10 in 34.4% of the cases. These long-term probabilistic dependencies cannot be observed in the model of Fig. 7 (or be discovered with any existing process discovery algorithm); however they showed up distinctly in our approach.

Results regarding *W_Beoordelen fraude (W_Be)*. Activity *W_Be* concerns the analysis of fraud in an application. For *W_Be* as core-activity, we discovered 34 patterns using the direct-context detector. Among those, pattern P11 (*W_Be* preceded by *A_PA*) stands out. If *W_Be* occurs in the context of P11, then only in 1.5% of the cases, the application was accepted (*O_AC*) and rejected in all other cases. If *W_Be* occurs in any other context, then 70.7% of the cases are accepted. In case of P11, *W_Be* is executed by the system whereas it is executed manually in all other cases. This suggests that the system-based fraud detection leads to rejection of possible fraud cases more rigorously than when fraud is decided manually.

Results regarding *O_AC*. The discovered model in Fig. 7 suggests that the four activities in part (b) can be executed in any order and also be skipped. We checked

this hypothesis and extracted patterns using the *concurrent detector*. For *O_AC* as core-activity, we find that *A_ACT*, *A_AP*, *A_RE* indeed occur concurrently in all 2243 cases (P-conf is 1.0), whereas the model allows skipping any of these. For *A_ACT* as core activity (pattern P12 in the following), *A_AP* and *A_RE* occur concurrently with confidence 1.0, but *O_AC* is *not* in 3 of 2246 cases. In these 3 anti-pattern instances of P12, *O_AC* is not executed at all, which may have severe financial impact according to the data owner: "*From a business point of view, this* [the cases where *O_AC* was skipped while *A_ACT* was executed] *implies that the customer never accepted an offer on a loan application, but the money was transferred nonetheless. In total, for 63,000 euro in these three cases.*"

Evaluation using an Insurance Log. For this evaluation we used a real-life insurance claims log which included 10,228 cases, 195,872 events spread over 18 event classes. A process model was obtained (see Fig 12) using the Inductive Miner [1] with default settings (noise threshold: 0.2). The model shows a flower-like behavior. Thus, the allowed behaviors of the model are general and hide specific behaviors.

Next we applied one of the built-in pattern detector (the Episode Detector) which returned 22 patterns, each of which involved the following activities: *conduct file review*, *incoming correspondence* and *follow up requested*. Twelve of the patterns had *incoming correspondence* as the core event and 9 of the patterns had *conduct file review* as the core event. Each of the three activities appeared as predecessor(s), successor(s) and concurrent with the core event across the detected pattern set.

Fig. 11: Snippet of Log Pattern Explorer visualization showing colors applied to activities identified in patterns returned by the Episode Detector.

By coloring these three tasks in the Log Pattern Explorer visualization, we can see that these tasks can happen almost in any order and at any point in time in a process (see Fig 11). We also note that, collectively, these three activities make up 68.5% of the total activities in the log. The relative positions of these three activities are shown in the discovered model. It is interesting to note that the activity *conduct file review* is not part of the flower structure in the discovered model but is part of a separate loop structure.

The LPE visualization provides an unfolded view of traces which clearly shows the interspersion of these three tasks across all cases. This facet of behavior will be hidden in a process model (or at least, it will not be obvious to those who are not trained to interpret process models carefully). Regardless, the magnitude of the interspersion will not be revealed by the simple process model visualization.

The visualization of the spread of these activities strongly suggests that these activities do not have any strong temporal relationships with any specific activities in the process. In other words, these activities are likely to be auxiliary activities that support the main process but are not temporally dependent on any other core claim process

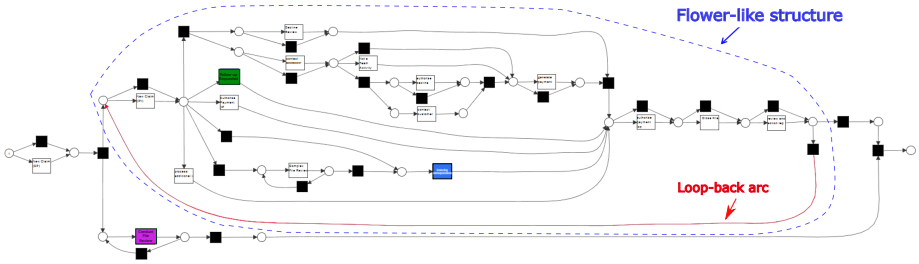

Fig. 12: Original process model discovered using Inductive Miner. The flower-like structure is outlined in blue dashed line, the arc indicating that the activities in the flower may be executed any number of times is shown in red. Corresponding colors have been applied to highlight the activities identified by the Episode Detector. Silent transitions are indicated by black boxes.

activities. In order to discover the main insurance process flow, we decided to remove these three auxiliary activities from the log to obtain the core process activities (though they may be relevant, for instance when analyzing performance).

Removal of auxiliary activities After filtering the activities *conduct file review*, *incoming correspondence*, and *follow up requested*, we sought to find behaviors representative of core insurance processes. We did so by (i) sorting the activities by frequency, and (ii) coloring the (top 10) most frequent activities accordingly. Furthermore, we also color the two *New Claim (IPI)* and *New Claim (CP)* activities as they seem to be referring to two types of claims and often occurred at the beginning of each case. This coloring exercise provides an interesting view: the distribution of activities between those cases that were (generally) started with *New Claim (IPI)* and *New Claim (CP)* as shown in Fig 13 and Fig 14 is quite distinct. Armed with this newly-gained insight, we split the log into *Variant 1* (those with activity *New Claim (IPI)*), *Variant 2* (those with activity *New Claim (CP)*), and *Variant 3* (the remaining cases).

Results The final process models that we obtained for each of the three variants exhibit significantly-reduced flower behaviors. Using the Projected Recall and Precision 3-activities metric [19]: the precision measure of the process model has increased from 0.47 (for the process model obtained after removing auxiliary events) to $>$0.7 (*Variant*

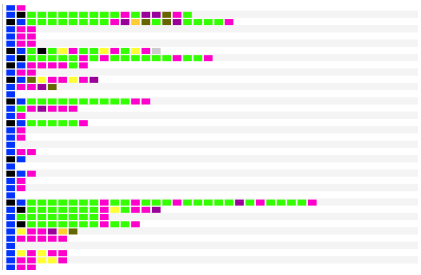

Fig. 13: Distribution of activities for cases with *New Claim (CP)* activity.

Fig. 14: Distribution of activities for cases with *New Claim (IPI)* activity.

Table 2: Comparison of related pattern detection approaches: (+) can detect; (+/-) may detect with suitable parameters; (-) cannot detect

	P0	P1	P3	P9a	P9b	P12	P12(anti)
Pattern abstraction [3]	+	+	-	-	-	+/-	-
Episodes miner [7]	+	+/-	-	+/-	-	+/-	-
Pattern discovery [8]	+	+/-	-	+/-	-	+/-	-
Local models [11]	+	+	-	+/-	-	+/-	-
Log pattern explorer	+	+	+	+	+	+	+

1), >0.9 (*Variant 2*), and >0.5 (*Variant 3*); the precision measure of the original model shown in Fig. 12 is 0.46.

Comparison to Unsupervised Approaches. Next, we compared our pattern detection approach to existing detection techniques regarding their ability to discover particular kinds of patterns. From the patterns discussed in the first case study, we tried to discover the 7 patterns listed in Tab. 2 using the 4 different unsupervised detection techniques [3, 7, 8, 11] discussed in Sect. 2; The frequent pattern P0 (sequence of activities) in Fig. 5 could be identified by all techniques; the infrequent P1 (concurrent activities) in Fig. 5 can be detected by unsupervised techniques only when lowering threshold parameters (which usually leads to very large result sets); the extremely infrequent pattern P3 (C-cover < 0.005) in Fig. 8 can not be detected due to for example implementation limitations (e.g., the tool of [7] requires frequency (C-cover) ≥ 0.05; the tool of [11] computes 500 patterns maximally). P9a (combines directly- and eventually-follows relations) in Fig. 10 cannot be detected by [3] (only directly-follows relations); the other techniques may detect it but do not distinguish directly and eventually-follows (only eventually-follows). P9b (combines directly-follows, eventually-follows and concurrency) in Fig. 10 cannot be detected by existing techniques as these would have to enumerate the possible interleavings of the pattern activities with all other process activities to find the pattern. For P12anti, no existing unsupervised approach supports detecting anti-pattern instances. Overall, existing approaches have difficulties with detecting patterns that are infrequent or contain a large set of concurrent (independent) events. The other patterns detected by these approaches can be converted into the *log patterns* as discussed in Sect. 5 and be further explored in the LPE tool.

7 Conclusion and Future work

In this paper, we proposed a semi-supervised approach for log pattern detection. We defined our patterns as partial orders and distinguished a core-activity to help the user detect patterns of interest and count instances of patterns. We use concurrency and contextual information of the core-events and support the user in extracting, modifying, and extending patterns. The two case studies show that the proposed approach helps to (1) detect complex patterns and infrequent patterns of interest, (2) modify/explore patterns (detected by existing unsupervised approaches or by our approach) and (3) gain insights into the parts of discovered models that are unstructured or over-generalized. However, in contrast to fully automated detection, some manual efforts are needed in our approach. Moreover, in the log visualization, an overview of the overall behavior of

the log is missing. A more extensive user study is needed to evaluate and improve the effectiveness of the approach. Future work aims at empirically evaluating the approach and the tool with more involvement of users and domain experts. Moreover, we would like to integrate log cleaning operations, such as event abstraction, event relabeling, event filtering etc., and recommend such operations for the patterns detected.

References

1. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - A constructive approach. In: Application and Theory of Petri Nets and Concurrency, 2013. Proceedings. (2013) 311–329
2. Monroe, M., Lan, R., Lee, H., Plaisant, C., Shneiderman, B.: Temporal event sequence simplification. IEEE Trans. Vis. Comput. Graph. **19**(12) (2013) 2227–2236
3. Bose, R.J.C., van der Aalst, W.M.P.: Abstractions in process mining: A taxonomy of patterns. In: BPM. Volume 5701 of LNCS., Springer (2009) 159–175
4. Günther, C., Rozinat, A., van der Aalst W.M.P.: Activity mining by global trace segmentation. In: BPM. Volume 43 of LNBIP., Springer (2009) 128–139
5. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P., Toussaint, P.J.: From low-level events to activities - A pattern-based approach. In: BPM. Volume 9850 of LNCS., Springer (2016) 125–141
6. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: An approach based on colored automata. In: BPM 2011. (2011) 132–147
7. Leemans, M., van der Aalst, W.M.P.: Discovery of frequent episodes in event logs. In: SIMPDA. Volume 1293 of CEUR., CEUR-WS.org (2014) 31–45
8. Diamantini, C., Genga, L., Potena, D.: Behavioral process mining for unstructured processes. J. Intell. Inf. Syst. **47**(1) (2016) 5–32
9. Suriadi, S., Andrews, R., ter Hofstede, A.H., Wynn, M.T.: Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. Information Systems **64** (2017) 132–150
10. Ferreira, D.R., Szimanski, F., Ralha, C.G.: Improving process models by mining mappings of low-level events to high-level activities. J. Intell. Inf. Syst. **43**(2) (2014) 379–407
11. Tax, N., Sidorova, N., Haakma, R., van der Aalst, W.M.P.: Mining local process models. J. Innovation in Digital Ecosystems **3**(2) (2016) 183–196
12. Baier, T., Rogge-Solti, A., Mendling, J., Weske, M.: Matching of events and activities: an approach based on behavioral constraint satisfaction. In: SAC, ACM (2015) 1225–1230
13. Song, M., van der Aalst, W.M.: Supporting process mining by showing events at a glance. In: Proceedings of WITS. (2007) 139–145
14. Lu, X., Fahland, D., van der Aalst, W.M.: Conformance checking based on partially ordered event data. In: BPM Workshops. Volume 202 of LNBIP., Springer (2014) 75–88
15. Ponce de León, H., Rodríguez, C., Carmona, J., Heljanko, K., Haar, S.: Unfolding-based process discovery. In: ATVA. Volume 9364 of LNCS., Springer (2015) 31–47
16. Mokhov, A., Carmona, J., Beaumont, J.: Mining conditional partial order graphs from event logs. T. Petri Nets and Other Models of Concurrency **11** (2016) 114–136
17. Diamantini, C., Genga, L., Potena, D., van der Aalst, W.M.P.: Building instance graphs for highly variable processes. Expert Syst. Appl. **59** (2016) 101–118
18. Lu, X., et. al.: Semi-supervised log pattern detection and exploration using event concurrence and contextual information (extended version). In: BPM Center Report BPM-17-01. (2017)
19. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Scalable process discovery and conformance checking. Software & Systems Modeling (Jul 2016)